Generating and Exploiting Deep Learning Variants to Increase Heterogeneous Resource Utilization in the NVIDIA Xavier

Roger Pujol
Universitat Politecnica de Catalunya (UPC), Spain
Barcelona Supercomputing Center (BSC), Spain

Hamid Tabani
Barcelona Supercomputing Center (BSC), Spain

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC), Spain

Enrico Mezzetti
Barcelona Supercomputing Center (BSC), Spain

Jaume Abella
Barcelona Supercomputing Center (BSC), Spain

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC), Spain

Abstract

Deep learning-based solutions and, in particular, deep neural networks (DNNs) are at the heart of several functionalities in critical-real time embedded systems (CRTES) from vision-based perception (object detection and tracking) systems to trajectory planning. As a result, several DNN instances simultaneously run at any time on the same computing platform. However, while modern GPUs offer a variety of computing elements (e.g. CPUs, GPUs, and specific accelerators) in which those DNN tasks can be executed depending on their computational requirements and temporal constraints, current DNNs are mainly programmed to exploit one of them, namely, regular cores in the GPU. This creates resource imbalance and under-utilization of GPU resources when executing several DNN instances, causing an increase in DNN tasks' execution time requirements. In this paper, (a) we develop different variants (implementations) of well-known DNN libraries used in the Apollo Autonomous Driving (AD) software for each of the computing elements of the latest NVIDIA Xavier SoC. Each variant can be configured to balance resource requirements and performance: the regular CPU core implementation that can run on 2, 4, and 6 cores; the GPU regular and Tensor core variants that can run in 4 or 8 GPU's Streaming Multiprocessors (SM); and 1 or 2 NVIDIA's Deep Learning Accelerators (NVDLA); (b) we show that each particular variant/configuration offers a different resource utilization/performance point; finally, (c) we show how those heterogeneous computing elements can be exploited by a static scheduler to sustain the execution of multiple and diverse DNN variants on the same platform.

2012 ACM Subject Classification Computer systems organization → Neural networks; Computer systems organization → System on a chip; Computing methodologies → Graphics processors

Keywords and phrases Deep Neural Network (DNN), GPU, Heterogenous Resources

Funding This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), and the HiPEAC Network of Excellence. MINECO partially supported Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717), Enrico Mezzetti under Juan de la Cierva-Incorporación postdoctoral fellowship (IJCI-2016-27396), and Leonidas Kosmidis under Juan de la Cierva-Formación postdoctoral fellowship (FJCI-2017-34095).

1 Introduction

Motivated by the higher accuracy achieved by deep learning (DL) solutions over traditional algorithms based on shallow learning, the use of DL in the mainstream computing domain has rapidly widespread. This covers a variety of areas ranging from pattern recognition to natural language processing. Critical real-time embedded systems (CRTES) are not an exception to this trend, with DL-based algorithms used in areas like robotics and autonomous driving (AD). In fact, DL has emerged as the reference algorithmic solution for the realization of several functionalities in AD such as computer vision (e.g., object detection and tracking), path planning, driver-monitoring systems, and voice-based command and control [11].

The other side of the coin is that DL increased accuracy comes at the cost of a substantial increase in the required computational demands. At software level, highly optimized frameworks, tools, and low-level libraries are deployed to improve the hardware utilization significantly, and also to facilitate the software development process [1, 26, 13, 27]. At hardware level, high-performance hardware is used to satisfy the massive computation needs of DL workloads [5, 2, 4], with GPUs at the forefront of those solutions and being extensively evaluated by OEMs and TIER companies in the automotive domain [3]. Despite these efforts, autonomous driving – the target domain of our work – still challenges the computational capabilities of existing solutions. Just for advanced driver-assistance systems (ADAS), which arguably require much lower performance than AD, ARM projects an 100x increase in computation needs from 2016 to 2024 [17]. Capturing these demands requires a computation capacity of tens of tera operations per second (TOPS), which can theoretically be achieved by having a variety of specialized computing elements (accelerators) in the GPU platforms and automotive system-on-chips (SoC), e.g., Tensor cores and deep learning accelerators.

Problem statement. Since DL is used in a variety of modules for different AD functionalities, several DL instances will be running simultaneously on the underlying SoC. For instance, while the object detector module analyzes the current frame, the tracking module processes the objects recognized in previous frames and matches them with the objects in the current frame. At the same time, the planning module calculates the best path trajectory. To make things worse, (i) each module can require several DNN instances to implement the required functionality, and (ii) the module can be instantiated several times, once per each input sensor, e.g. camera, LiDAR, and radar. However, while modern GPUs offer a powerful heterogeneous platform with several type of (accelerating) computing elements (CE), current DL libraries are implemented to mostly exploit one of them, which at the time of writing this paper are the regular cores in a GPU. Regardless of the specific CE, the fact that just one CE is used, heavily under-exploits modern heterogeneous SoC computation capacity. Our view is that the ability to run DL-based *variants*, each using different CEs, would improve timing and throughput, and would pay off the extra effort required to implement those different variants. As a matter of fact, recently, NVIDIA integrated powerful deep learning accelerators (NVDLA) designed and specialized for DL workloads, and Tensor cores for DL inference, which are capable of different data type operations, from `int8` to `fp16` and `fp32`, and provide massive and flexible computation capacity.

Contributions. In this paper, with focus on the Xavier SoC and the Apollo AD framework [9], we make the following main contributions:

1. DNN usage in Apollo. We perform an analysis of the number of DNN instances that can be active during the execution of Apollo. We show that at least seven instances can be active at the same time and each instance comes with different computation needs and different time constraints. Also, based on observed indicators, we conclude that the

number of DNN instances is expected to increase in future AD systems.

2. We implement distinct variants of different DL libraries so that each DL application can be executed on different CEs in the NVIDIA's Xavier SoC: CPU, GPU regular cores, GPU Tensor cores, and NVDLA. Our variants are programmed such that they can be executed under different thread-level parallelism (TLP) degrees. This allows more flexibility when exploiting the existing CEs.
3. We make an in-depth analysis of the implications of running the different variants of the DL libraries on the NVIDIA's Xavier SoC and show that each implementation/TLP-setup offers a different design point in terms of used resources and performance.
4. We model a multicore cyclic executive scheduler as a linear programming (LP) problem to assess the increase in guaranteed performance enabled by heterogeneous resources. We show how the variable execution requirements exhibited by tasks on the different heterogeneous computing elements can be exploited to increase the number of advanced neural network based functionalities on the same SoC, with clear advantages in terms of reduction in procurement costs and reliability concerns.

The rest of this paper is structured as follows. Section 2 introduces DNNs and Apollo. Section 3 analyzes the DNNs used in Apollo and the projection in the use of DNN in CRTES. Section 4 presents the main details of our target platform, the NVIDIA's Xavier SoC. Section 5 details the different implementations we developed for different DL libraries and their resource usage and performance in the Xavier SoC. Section 6 shows how scheduling can benefit from this TLP-configurable implementations to increase system load or adapt DL execution time requirements to its allocated time budget. Section 7 presents the most relevant related works, and, finally, Section 8 summarizes the main conclusions of this work.

2 Deep Neural Networks and their use in AD

2.1 Introduction to DNN

Deep Neural Networks (DNNs) [35] provide high accuracy solutions in several domains including computer vision for functions such as image classification and object detection. Nowadays, DNNs are widely used in a variety of areas and CRTES are not an exception to this. Recurrent neural networks (RNNs) are another class of artificial neural networks with internal state that are very successful for history-based workloads such as speech recognition, path planning and machine translation. RNNs are used as the state-of-the-art approach for path planning in industrial autonomous driving systems.

2.2 Apollo Autonomous Driving Software

We study *Apollo* [9], arguably the most sophisticated open-source autonomous driving framework available and already deployed on a variety of prototype vehicles (including autonomous trucks). Apollo supports state-of-the-art hardware such as latest LIDARs and cameras from Velodyne and other vendors, as well as GPU acceleration. Apollo comprises 8 main modules and several sub-modules, as shown in Figure 1. These software modules operate in a software-pipelined fashion and work in general at frame level.

M_0 *Speech recognizer* processes the voice-based commands from the driver/passengers and transmit them to the control unit.

M_1 *Perception* identifies the surrounding area around the autonomous car.

$M_1.d$ The *detection* submodule is in charge of detecting obstacles and objects from different sensors.

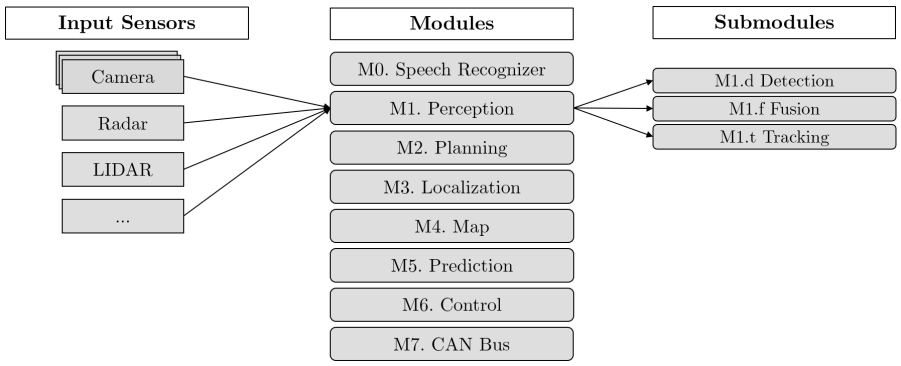


Figure 1 Modules, sub-modules and input sensors of Apollo.

$M_1.f$ *fusion* takes the results of all detected objects from different sensors and combines them by a sensor fusion algorithm.

$M_1.t$ *tracker* follows the detected objects and matches them with the objects detected in previous frames.

M_2 The *Planning* plans the spatio-temporal trajectory for the vehicle to take.

M_3 *Localization* leverages information received from different input sensors to estimate the precise position of the vehicle.

M_4 The *Map* provides ad-hoc structured information regarding the roads.

M_5 *Prediction* anticipates the future motion trajectories of perceived obstacles/objects.

M_6 *Control* generates control commands such as accelerating/braking and steering.

M_7 *CAN Bus* passes all the control commands to the vehicle hardware and provides information back to the autonomous system.

In this paper, we used Apollo default input data sets which are real data from sensors of an AD car collected and provided by the Apollo team. In addition, we used similar neural network architectures that Apollo employs in its different stages.

3 Analysis of the DL elements in Apollo

In the literature, we can find a wide range of DNNs and other DL algorithms. In this paper, we focus on those normally used for DL-based solutions in AD systems. In this line, Table 1 shows different types of (state-of-the-art) neural networks widely used in key domains for AD functionalities. We can observe that three modules (M_0, M_1, and M_5) and 2 sub-modules ($M1.d$ and $M1.t$) use neural networks, DNNs and RNNs in particular. Table 1 shows:

- The perception module, M_1, relies on different DNNs for detecting ($M1.d$) obstacles and objects from different sensors. The results of all detected objects are fused by a sensor fusion algorithm ($M1.f$) that does not use DNNs. As a last step, an object tracker ($M1.t$) deploys a DNN to track and follow detected objects.
- The prediction module, M_5, uses RNNs to build a model to predict the target lane that the vehicle should take. One RNN model is for lane sequences and another RNN model for the associated object states. The concatenation of these two RNNs is fed into another neural network to estimate the probability for each lane sequence. Interestingly, the modules using neural networks, *Perception* and *Prediction*, are the most compute-intensive modules: they consume more than 70% of the time Apollo uses to process each frame.

Table 1 Neural networks used in different modules of the Apollo autonomous driving system.

	Deep Learning Software	Description
M0. Speech Recognition	Voice Command and Control	A DNN-based accurate speech recognition application to process speech commands
M1. Perception	Camera Object Detection	A DNN-based algorithm to identify objects and traffic signals from camera sensors
	LiDAR object Detection	A DNN-based algorithm to identify objects from LiDAR sensors
	Object Tracker	A DNN-based algorithm to track identified objects in consecutive frames
M5. Prediction	Lane sequences (RNN1)	A RNN for lane sequence-based prediction
	Obstacle status (RNN2)	A RNN for obstacle status
	A RNN using the output produced by RNN1 and RNN2	A RNN to compute the probability of each lane sequence based on RNN1 and RNN2

- Some AD systems suggest to deploy AI-assistant applications to be implemented inside the cabin, which are all based on neural networks [11]. Such applications are proposed for driver-monitoring, and command and control using gestures and voice. and RNNs.

Table 2 summarizes the modules using DNNs and RNNs.

Table 2 Modules of Apollo using DNNs (⊛) and RNNs (◎).

	M1 (Perception)			Other Modules						
Input	M1.d	M1.f	M1.t	M0	M2	M3	M4	M5	M6	M7
Camera	⊛									
LiDAR	⊛		⊛	⊛				◎ ◎ ◎		
Radar										

3.1 Real Execution Trace

Figure 2 shows a trace collected from an actual execution of Apollo when all DNN and RNN instances in the different modules/sub-modules use the regular GPU cores in the Jetson AGX Xavier. Each rectangle shows the span of execution of each DNN/RNN instance, i.e. since it starts running (i.e. it is executable) until its execution finishes.

As Figure 2 shows, at a given time t_i, several instances of different neural networks are executed concurrently. Also, each particular DNN/RNN has diverse computational requirements with more than 12× variability among them: the voice command runs for 4.5ms, whereas the different RNNs in the prediction module range from 48.61ms to 61.06ms; finally in the perception module DNN instances span goes from 38.96ms to 50.18ms. It is also the case that temporal constraints vary up to 10x across DNN/RNN instances. This comes from the fact that the rate at which frames arrive across different input sensors can vary from 10ms for Radars to 100ms for LiDARs.

3.2 DNN instances

Current trends show that the number of concurrent neural network instances can easily reach dozens:

- *More input sensors.* Moving toward fully autonomous driving (Level 5 [8]) will naturally require to increase the number of sensors to cover the car's surrounding more accurately. Today, some of the AD systems, which are still far from a Level 5 system, use more than 8 cameras and radars (e.g., Telsa [28] uses 8 cameras and 12 ultrasonic sensors, and

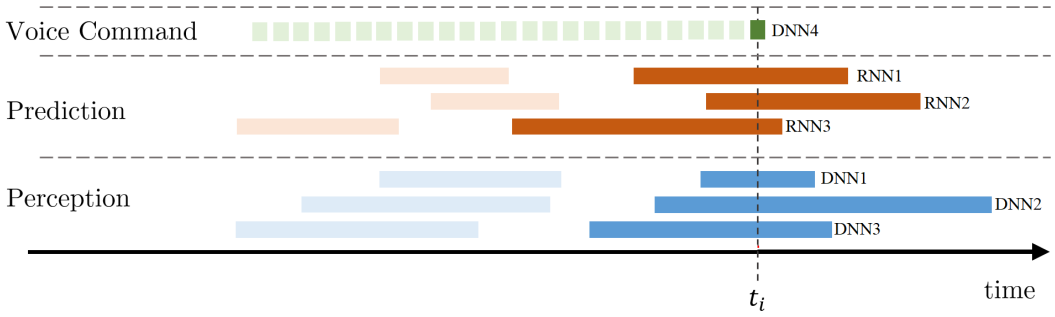


Figure 2 Concurrent execution of different modules or instances of a module in Apollo, an industrial autonomous driving software.

NVIDIA autopilot [11] uses 8 high-resolution cameras, 8 radars and optionally up to 3 LiDARs). Therefore, more DNN-based workloads will have to be processed, increasing the computation demand significantly.

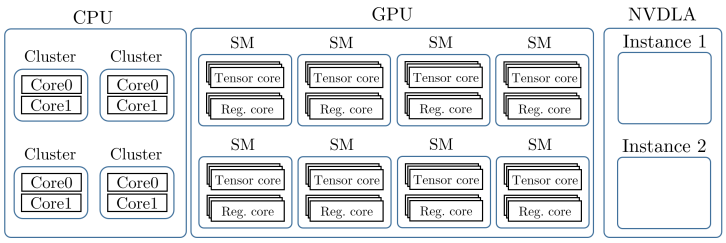
- *More sophisticated algorithms.* Perception submodules tend to use more sophisticated DNNs, with larger number of layers and, therefore, higher computational needs for further improvements in the accuracy of object and obstacle detection, specially in conditions with reduced visibility such as fog, night, rain, and snow. The *Prediction* module already uses 3 different neural networks either to achieve higher accuracy or to cover more complex scenarios. Indeed, this type of modules normally use sophisticated neural network architectures [44, 45].
- *More functionalities.* Besides the main functions of an AD system, extra features are introduced to improve the quality and safety of driving: from gesture detection and speech-based command and control up to driver-monitoring to predict take-over readiness [30].

This trend towards exploiting multiple neural networks running in parallel and the increasing number and type of accelerators we witness in modern GPUs motivate our idea of assessing the benefits of DNN/RNN variants in modern GPUs.

4 Main Computing Elements in The Jetson AGX Xavier

NVIDIA has recently introduced the Xavier SoC as the cornerstone of its automotive platforms. Xavier delivers over 30 TOPS for DL applications while consuming less than 30 Watts. Xavier comprises four main computing elements (CEs) capable of processing deep learning workloads: traditional CPU cores, GPU cores, GPU Tensor cores, and the NVDLAs. The Xavier SoC also integrates several other accelerators such as vision accelerator, video encoder, etc. However, these accelerators cannot be used for DNN/RNN inference, due to their limited programmability. Therefore, in this paper, we focus only on CPU, GPU (regular and Tensor) cores, and the NVDLA.

1. CPU cores. The CPU complex (CCPLEX) comprises eight homogeneous carmel ARMv8.2 processors. Each core has its private instruction and data caches. In each cluster of two cores (4 clusters in total), an L2 cache is shared between both cores. An L3 cache is shared between all CPU cores.
2. GPU regular and Tensor cores. The Volta GPU microarchitecture comprises 512 regular cores (CUDA cores in NVIDIA terminology) and 64 Tensor cores. The GPU is structured in 8 Streaming Multiprocessors (SMs) each containing 64 regular and 8 Tensor cores.



CPU	GPU	NVDLA
1 cluster	4 SMs	1 instance
2 clusters	8 SMs	2 instances
3 clusters		

Figure 3 CEs in the Xavier SoC and granularity at which we exploit them.

Tensor cores [12] accelerate large matrix operations, which are at the heart of many AI functions. While each regular core can perform up to one single precision multiply-accumulate operation per 1 GPU clock, each Tensor core can perform one matrix multiply-accumulate operation per 1 GPU clock. The Tensor core can multiply two `fp16` 4×4 matrices and adds the multiplication product `fp32` matrix to the accumulator, which is also a `fp32` 4×4 matrix.

In each SM, threads can use either the regular cores or the Tensor cores. Hence, at most, 512 regular or 64 Tensor cores can be used in parallel.

3. NVDLA provides a flexible, robust inference acceleration solution. Xavier SoC has two NVDLAs which can be configured to run deep learning workloads. To the best of our knowledge, this is the very first work that considers NVDLAs in the real-time domain.

Overall, the NVIDIA Xavier SoC offers four different CEs that we will use to illustrate the benefits of our proposal. In order to reduce the exploration space we reduce the granularity at which we explore each CE, see the table on the right in Figure 3.

- At the CCPLEX level, we restrict our approach to core clusters. Also, since all DNN/RNN instances using the GPU or the NVDLA are initiated from the CPU, we reserve 2 CPUs for them. Overall, at the CPU level, a DNN/RNN instance can use 2, 4, or 6 of the remaining cores.
- At the GPU level, we setup a minimum granularity of 4 SMs. Hence, a DNN/RNN task can use either 4/8 SMs to exploit 256/512 GPU regular or 32/64 tensor cores, respectively.
- Each task can use one or two NVDLA accelerators.

5 Diverse DNN Implementations

This work started with a massive effort to port Apollo to the Jetson AGX Xavier. In fact, to our knowledge, this is the first work showing results of Apollo industrial framework on NVIDIA GPUs. To that end, we change the implementation in the baseline source code which is based on x86 and GPU. Depending on the target CE, we need to use appropriate libraries and re-implement Apollo modules. Deep learning workloads, in general, are implemented layer by layer by defining specific functions for each layer. Then, depending on the layer and on the highly-optimized low-level target library (e.g., cuBLAS) input data needs to be transformed to match the proper format expected by the low-level library function.

The current version of Apollo exploits only regular cores in the GPU for inferencing DNNs. The most computationally-intensive part of inference, such as convolution or fully connected layers, are usually reduced to GEneral Matrix Multiplication (GEMM), which are

implemented with cuBLAS [1]. It is worth mentioning that, to our knowledge, the version of Apollo that we studied in this paper does not use TensorRT.

5.1 Specialized per-CE libraries

Table 3 presents the optimized libraries that we have used to implement our software. As it can be seen, for each specific CE, we used different libraries. In addition, we modified the baseline code in order to run the optimized code. Recently, as part of the introduction of Tensor cores, NVIDIA provided some low-level libraries to support their use. NVDLA, which can be accessed through TensorRT [6], is a platform for high-performance deep learning inference. TensorRT offers a deep learning inference optimizer and runtime that can deliver low latency and high-throughput for DL inference applications.

Table 3 Optimized libraries used to implement the Apollo software for each particular CE.

CE	Optimized Libraries
CPU	We used openMP [29] to implement all the functions to run on the CPU cores. Our implementation allows fixing the maximum number of cores that can be used.
GPU Regular Cores	The baseline implementation targets regular cores to run the kernels.
GPU Tensor Cores	We used specific libraries and adapted our code to exploit the Tensor cores. Some of our target deep neural networks consist of 100+ layers. The implementations of all the layers had to be modified.
NVDLA	We adapted each neural network configuration to be compatible with TensorRT [6], except the RNNs as they are not supported by NVDLA [7]. We use the TensorRT framework to launch applications on the NVDLAs.

5.2 Implementation for different CEs

We illustrate the required effort to modify all the functions in the source code to run the entire workload on a specific CE, by focusing on a small function performing a matrix multiplication (GEMM) operation without transposing the operand matrices. It is worth noting that each of the functions that implement the different layers of the neural networks are functionally different and therefore, each of them requires different modifications.

In this example, the matrix multiplication function builds on the following formulation, in which A, B, and C are matrices and α and β are floating-point coefficients.

$$C = \alpha A \times B + \beta C \tag{1}$$

5.2.1 CPU implementation

Figure 4 shows the CPU version of the matrix multiplication operation presented in Equation 1. As input parameters the function takes $ALPHA$ and $BETA$ as shown in Equation 1; M, N, and K that are the dimensions of the matrices; and lda, ldb, and ldc are leading dimensions of matrices A, B, and C respectively. In other words, lda, ldb, and ldc determine the forward move in memory when is reached the end of a row (in row-major order) or column (in column

[1] For completeness, we have performed several experiments comparing the same DNN operations using cuDNN and cuBLAS. Our results show that cuBLAS achieves very competitive results w.r.t. cuDNN. However, note that main idea of the paper, i.e. having diverse DNN implementations, does not depend on the particular library used.

major). In fact, these parameters define strides that provide plenty of flexibility to work with smaller tile sizes inside a larger matrix.

In the first loop, lines 4-8, the βC operation is executed according to Equation 1. In lines 10-17, the main loops are implemented to perform the matrix operations. The openMP pragma at line 9 will automatically parallelize the outer loop so that independent loop iterations can be executed in parallel.

```
1  void OpenMPgemmNN(int M, int N, int K, float ALPHA, float const *A, int lda, float
       const *B, int ldb, float BETA, float *C, int ldc)
2  {
3      int i, j,k;
4      for(i = 0; i < M; ++i){
5          for(j = 0; j < N; ++j){
6              C[i*ldc + j] *= BETA;
7          }
8      }
9      #pragma omp parallel for
10     for(i = 0; i < M; ++i){
11         for(k = 0; k < K; ++k){
12             register float A_PART = ALPHA*A[i*lda+k];
13             for(j = 0; j < N; ++j){
14                 C[i*ldc+j] += A_PART*B[k*ldb+j];
15             }
16         }
17     }
18 }
```

Figure 4 CPU implementation of the reference matrix multiplication (gemm) operation.

5.2.2 GPU regular core implementation

Figure 5 shows the implementation for the GPU regular cores, with the function requiring the same parameters as for CPU version. Also note that in this example, we assume that the matrices are already in the device's memory space.

```
1  void GRCSgemmNN(int M, int N, int K, float ALPHA, float const *A,
2  int lda, float const *B, int ldb, float BETA, float *C, int ldc)
3  {
4      static int init[16] = {0};                  // Vector for initialized handles
5      static cublasHandle_t handle[16];           // Vector of actual handles
6      int i;
7      cudaGetDevice(&i);                          // Get current device
8      if(!init[i]) {                              // If not initialized
9          cublasCreate(&handle[i]);               // Creates the handle
10         init[i] = 1;
11     }
12     cudaError_t status = cublasSgemm(handle[i],
13             CUBLAS_OP_N, CUBLAS_OP_N,           // Select the non-transpose matrices
14             N, M, K,                            // Sizes of the matrices
15             &ALPHA,
16             B, ldb,                             // B and it's leading size
17             A, lda,                             // A and it's leading size
18             &BETA,
19             C, ldc);                            // C and it's leading size
20     if (status != cudaSuccess)                  // Check if there is any error
21         printf("CUDA Error: %s\n",cudaGetErrorString(status));
22 }
```

Figure 5 GPU implementation for regular cores.

First, we get the device ID and we check whether we have initialized a cuBLAS handle for it. If it is not the case we create a new one. Once we obtain the handle, we call `cublasSgemm` but with the matrices in reversed order. This is because C/C++ assumes a row major

layout whereas CUDA assumes column major layout, which means that CUDA is reading the matrices in a transposed manner. Then, since everything is transposed, we can simply reverse the operators:

$$A \times B = C \iff B' \times A' = C'$$

Finally, we check whether cuBLAS triggered any error during the GEMM.

5.2.3 GPU Tensor core implementation

Reprogramming the GPU code to be run on the Tensor cores only requires to change the math mode to `CUBLAS_TENSOR_OP_MATH`. Nonetheless, this implementation builds on some preconditions to run on the Tensor cores: K, lda, ldb and ldc have to be multiple of 8 and N has to be multiple of 4. Figure 6 shows the implementation for the GPU Tensor cores.

```
void GTCSgemmNN(int M, int N, int K, float ALPHA, float const *A, int lda, float
    const *B, int ldb, float BETA, float *C, int ldc)
{
    static int init[16] = {0};                  // Vector for initialized handles
    static cublasHandle_t handle[16];           // Vector of actual handles
    int i;
    cudaGetDevice(&i);                          // Get current device
    if(!init[i]) {                              // If not initialized
        cublasCreate(&handle[i]);               // Creates the handle
        init[i] = 1;
    }
    cublasSetMathMode(handle[i], CUBLAS_TENSOR_OP_MATH);     // Set math mode to
    enable Tensor cores
    cudaError_t status = cublasSgemm(handle[i],
            CUBLAS_OP_N, CUBLAS_OP_N,           // Select the non-transpose matrices
            N, M, K,                            // Sizes of the matrices
            &ALPHA,
            B, ldb,                             // B and it's leading size
            A, lda,                             // A and it's leading size
            &BETA,
            C, ldc);                            // C and it's leading size
    if (status != cudaSuccess)                  // Check if there is any error
        printf("CUDA Error: %s\n",cudaGetErrorString(status));
}
```

Figure 6 GPU implementation for Tensor cores.

5.2.4 NVDLA

The steps we have followed to run the neural network workload on the NVDLAs, are shown in Figure 7. As a first step, the DNN configuration needs to be in the proper format, *prototxt* that is compatible with TensorRT. To that end, we developed a script that goes layer by layer in the configuration file of the neural network and changes its format to *prototxt*. It is worth mentioning that some layers in the original format are translated into several layers in *prototxt*. For instance, a *Convolutional* layer that has *Batch Normalization* and an activation of type *Leaky* is divided into four different layers: a regular convolution, a *Batch Normalization*, a scale, and a *ReLU* (Rectified Linear Unit) with negative slope.

Following this step, we obtain a functional configuration file in the proper format. However, in most cases, some layers are not supported yet by TensorRT. At the time of writing this paper, several types of layers, especially for RNNs, are not implemented, and therefore, cannot be executed on the NVDLA. To overcome this limitation, we adapted some of the available layers using equivalent and currently supported methods. The conflicting layers in our neural networks are *Upsample* layer and *Leaky ReLU* layer. To solve this issue, we implemented the

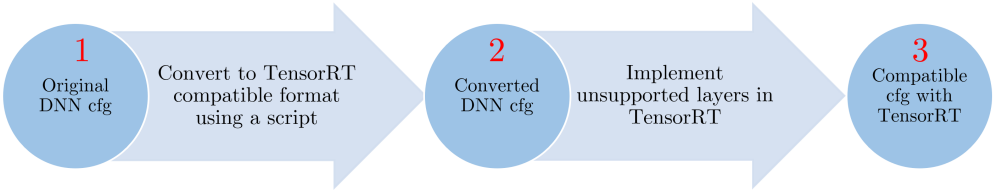


Figure 7 The steps required to specify neural network layers in order to be run on the NVDLAs.

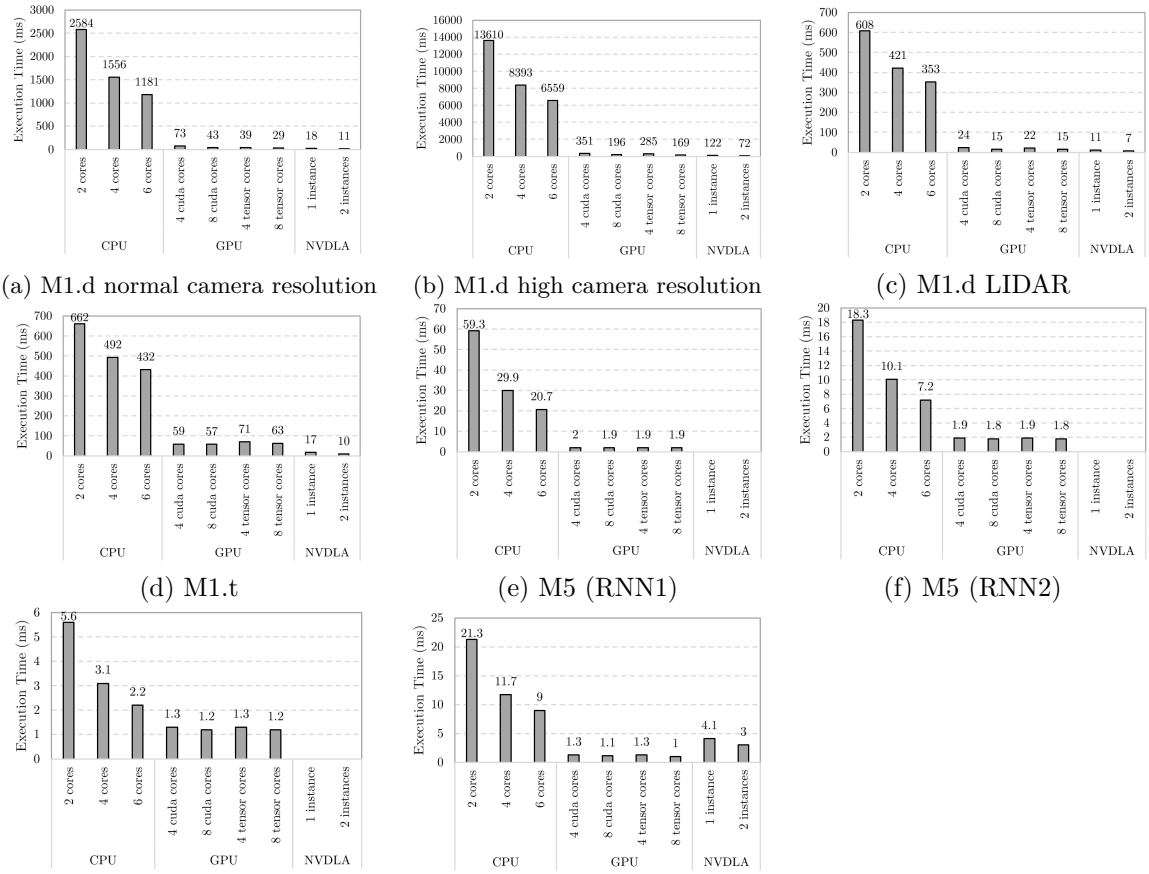
layers according to TensorRT specification. After this modifications, configuration files are in the correct format required by TensorRT. Using the generated configurations, TensorRT can parse and build the model of neural network and run it in the NVDLAs. Note that TensorRT can allow unimplemented layers to fallback and run on the GPU cores. In our experiments for this paper, we avoid any fallback by adapting all the unimplemented layers using other equivalent layers (this process does not include RNN layers which, as stated before, are not supported) and the entire neural network is running on the NVDLA.

5.3 Timing Analysis Results

We analyse the results obtained for each DNN variant under different CE and TLP levels: CPU cores (2 cores, 4 cores, and 6 cores), GPU regular cores (4 SMs, 8 SMs), GPU Tensor cores (4 SMs, 8 SMs), and NVDLAs (1 or 2 NVDLAs). Note that we always reserve 2 cores of the CPU for managing the operating system tasks and the tasks that are running in the GPU or NVDLA, since they are also triggered by their corresponding CPU processes.

Figure 8 shows the timing results of different neural network instances of Apollo running on each CE. Timing characterization has been performed with other DNN/RNN instances run in parallel. While we did not run specific experiments to hit the worst-case timing interference among computing elements, we assume the obtained results also factor in contention effects. We can derive the following conclusions.

- For the camera object detector ($M1.d$), Figure 8 (a) shows that using more CPU cores significantly improves the performance. Regarding the timings on GPU CEs, as expected for this workload, Tensor cores provide better performance in comparison to the regular cores, with 8 SMs providing significantly higher performance in comparison to 4 SMs. Also, the NVDLA accelerates this NN achieving the best performance results.
- Figure 8 (b) shows the results for ($M1.d$) under another configuration for the object detector with the same neural network architecture, but with a higher camera resolution. As the results show, we have the same trends as in Figure 8 (a), however, due to the increase in the workload size, execution times increase.
- Figure 8 (c) shows the results for the LiDAR object detector, $M_1.d$. Similarly to the previous results, GPU CEs provide better performance than CPU cores, though this time Tensor cores do not result in significant improvements over GPU regular cores. NVDLA again provides the best results. As the workload is smaller than for camera detector, the times are proportionally reduced.
- Figure 8 (d) shows the timing results for object tracker, $M_1.t$. For this specific workload, GPU regular cores provide higher performance than the Tensor cores. After a detailed analysis and designing some experiments, we find out that Tensor cores achieve worse performance than regular cores whenever we run a GEMM of $A_{M\times K}B_{K\times N}$, in which N has a very small value. More specifically as Figure 9(a) shows for $N \leq 12$ Tensor cores exhibit worse performance than regular cores. This particular case directly affects the



(a) M1.d normal camera resolution (b) M1.d high camera resolution (c) M1.d LIDAR

(d) M1.t (e) M5 (RNN1) (f) M5 (RNN2)

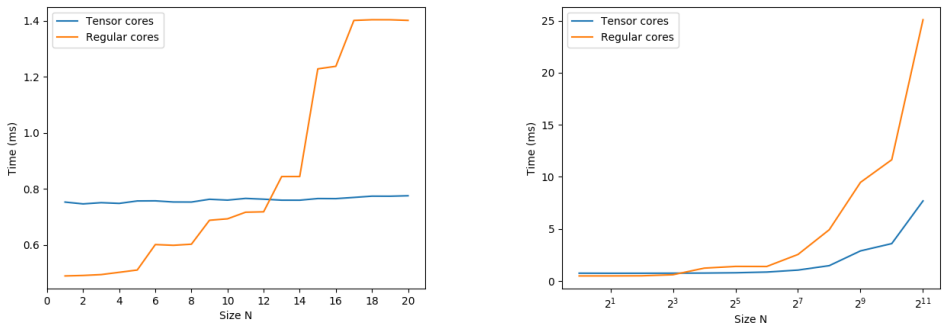
(g) M5 (RNN3) (h) M0 (Speech and comm. control)

Figure 8 Timing results of different Apollo neural network instances on different CEs.

Object Tracker (M1.t) since all the GEMMs that are performed by this DNN have $N = 1$. However, by increasing the value of N, (in this particular experiment, for $N > 12$) Tensor cores provide considerably better performance, see Figure 9 (b).

- Figures 8 (e), (f), and (g) show the timing results for the three recurrent neural networks in the prediction module. Due to the nature of the RNNs, these workloads cannot highly benefit from increased parallelization. Also GPUs provide up to an order of magnitude better performance in comparison with CPUs. However, this is too far from highly utilizing the GPU resources. In terms of the NVDLA, since several key layers of the RNN networks are not implemented in the TensorRT, we are unable to run these workloads on the NVDLA [10].
- Finally, Figure 8 (h) shows the speech recognizer module ($M0$) which uses a DNN as discussed in previous sections. This DNN network improves performance by one order of magnitude with GPU cores. Instead the NVDLA, while better than the CPU cores, performs significantly worse than the GPU cores.

Overall we can see that, (i) for some DNNs the NVDLA variant and the GPUrc (regular cores) and GPUtc (Tensor cores) variants offer comparable performance. (ii) Some times the GPUrc variant provides better results than GPUtc. (iii) It is also the case that in some cases the performance of CPU is relatively close ($\approx 2x$) of that obtained with the GPUtc and GPUrc variants. (iv) Across the different neural networks, we see that the CPU time



(a) N ranging from 1 to 20. **(b)** N ranging from 1 to 2^{11}.

Figure 9 Time spent in a GEMM ($A_{M\times K}B_{K\times N}$) where $M = K = 1024$.

requirements for some of them (e.g. (e), (f), (g), (h)) is comparable to that required by others in the GPU and NVDLA (e.g. (a), (b), (c), (d)). This makes it worth exploiting all CEs.

5.4 Other Considerations

TLP controllability. Fully exploiting variants requires exercising control on TLP as provided by NVIDIA's MPS (Multiprocessing Service), which allows multiple kernels from different processes to be executed concurrently in the GPU, while limiting their resource usage i.e., how many SMs each kernel will be using. The use of MPS has been shown to provide positive results in real-time systems [42], which paves the way for its ubiquitous adoption in all GPUs. Furthermore, MPS only requires driver updates. As this feature is not present in the Xavier SoC, we emulate its effect in our experiments by executing the GPU tasks in isolation and using the Xavier's capability to enable only a certain number of SMs in the GPU.

Contention effects on timing behavior is a widely studied topic in the real-time community mainly for CPUs, with few techniques proposed for GPUs [25] to reduce contention bounds. Contention bounding techniques, e.g. [38, 31] produce a factor Δ_{cont} to be added on top of the in-isolation timing estimates. In the scope of this paper we assume that the observed execution time factor in relevant contention effects.

Accuracy. Different implementations may use different standards for floating-point (FP) number representation (e.g. 16-bit or 32-bit representations), different FP operations or, at least, different FP operation orders. Due to rounding effects, this may lead to slightly different numerical results, whose impact on the system-level functionality needs to be assessed. However, functional results (i.e. objects detected, driving decisions, etc.) match since those tiny numerical variations have no impact in the semantics of the framework. For instance, whether the probability of recognizing an object varies by $\pm 0.1\%$ makes no practical difference in general (e.g. 90.7% vs 90.8%). Hence, despite the different implementations across CEs, the results of all implementations match functionally.

Multi-CE variants. In our current implementation, each neural network instance exploits a single CE. As future work, we consider adding a multi-CE capability, so that a single instance can exploit several CEs, e.g. 4 cores, 1 SM, and 1 NVDLA. While this offers more flexibility, our current single-CE per neural network instance approach already shows significant improvements over the baseline in which all instances use the same CE.

6 Exploiting Diversity to Increase Schedulability

With platforms supporting 'diverse' computing elements and TLP degrees, the timing behavior of an application is inherently dependent on the deployed configuration. Applications will exhibit different execution time bounds depending on the actual CE they are mapped to. The overall mapping strategy is thus fundamental to determine the schedulability of a given set of applications as a whole. The identification of optimal mapping strategy, is not a specific requirement for heterogeneous platforms [19], but is a well-studied problem at the basis of several scheduling approaches for homogeneous systems, from partitioned to cyclic-executive scheduling approaches (e.g., [36, 23]). Computing an optimal partitioning is NP-hard in the general case: depending on the complexity of the problem instance, provided solutions range from exact optimization frameworks to heuristic-based approaches.

In this paper, we are interested in assessing the benefits, in terms of system schedulability, that can be enjoyed with execution platforms supporting diverse CE/TLP configurations. As a common characteristic, the different DNN instances realizing the functionalities of the AD framework can be modeled as *recurrent applications* that are periodically executed according to a given frame rate. The frame rate depends on the frequency at which inputs need to be elaborated. Static scheduling or cyclic-executive approaches are particularly suitable for this kind of systems: despite their known limitations in term of flexibility and scalability, they are relatively easy to implement and provably predictable, even on multicores. For this reasons, cyclic-executive is still widely adopted in the critical embedded real-time system domains, and is at the basis of standard frameworks (e.g., AUTOSAR [18], ARINC [16]) in critical embedded real-time system domains.

A static schedule results in the repeated execution of a sequence of intervals or frames. Tasks associated to a frame must execute and complete within that frame (i.e., performance guarantees are enforced at each frame boundary). A sequence of frames is then periodically repeated as part of a major frame, corresponding to the hyperperiod. The recurrent behavior of the diverse DNN instances (and the relative independence between them) is naturally modeled with a static schedule. Constructing a schedule for a cyclic executive consists in finding a task-to-core mapping that allows all tasks to complete within their frame (or, reciprocally, that the cumulative utilization of all tasks in a frame does not exceed 1). While there exist specific rules to define appropriate frame number and size, the schedulability of a cyclic executive systems reduces to showing that all computations have completed within the frame. A common approach to construct a valid static schedules consists in formulating the scheduling problem as a *linear programming* (LP) model.

In the scope of our evaluation, we model the problem of scheduling an heterogeneous workload of several diverse DNN instances as a cyclic executive system. We exploit a LP-based representation of the problem to assess the increase in schedulability that can arise when multiple CE/TLP configurations are supported. Without loss of generality, we assume in this paper that all DNN/RNN variants share a common time frame: the ILP formulation allows intercepting those deployment scenarios where it is impossible to schedule all the DNN variant within a frame. In the following we first discuss our assumptions in terms of schedule constraints and LP formulation, and then we present the experimental set-up.

6.1 Task Model and Linear Programming Model

We consider a periodic task system $\mathcal{T}$ and we model DNN variants as a set of n independent periodic tasks $\tau_1, \ldots, \tau_n \in \mathcal{T}$ that have to be statically scheduled on a multiprocessor platform, comprising a set of heterogeneous m cores. We assume an implicit-deadline periodic task

model where each task τ_i is characterized by a period p_i and a relative deadline d_i (in this work we assume implicit-deadline tasks, thus $d_i = p_i$).

Given the heterogeneous nature of the platform, a task cannot be associated to a single-valued computational requirement. Moreover, it is not even sufficient to model the variation in the time as a function of the specific core the task is executing on. As an example, the GPU in the Xavier SoC include 8 Streaming Multiprocessors, which can be used as regular (CUDA) cores or can be configured to exploit also the Tensor cores, and an application (e.g., a DNN instance) may be executed on a variable number of SMs. Therefore, each task may exhibit different time bounds depending on the computational element (and mode) it is executed, and the TLP degree it is granted. We capture this dimensions as a set of CE/TLP configurations $\mathcal{CE} := \{ce_1, \ldots, ce_k\}$ so that each task is associated a set of time bound $\mathcal{C} = \{c_{i,1}, c_{i,2}, \ldots c_{i,k}\}$ with $c_{i,j}$ denoting the time of task τ_i under configuration $ce_j \in \mathcal{CE}$.

In line with our assessment objective, we are not interested in modeling a full static schedule over a full major frame. We limit our scope to the problem of finding a feasible schedule (if it exists) at the smallest time interval (frame) at which timing constraints are enforced. Given a set of tasks $\mathcal{T}' \subseteq \mathcal{T}$ to be statically scheduled on a set of CE/TLP configurations $\mathcal{CE}$ within a frame f, a static schedule for $\mathcal{T}'$ in f under configuration $ce_j \in \mathcal{CE}$ is valid only if the cumulative task utilization does not exceed f.

We modeled the cyclic executive scheduling problem on multiple CE/TLP configurations as an LP problem. LP-based approaches have been exploited for deriving static schedules in both homogeneous [23] and heterogeneous [19] multiprocessor systems. While the considered optimization problem is NP-hard [37], LP approaches have been shown to be effective in most cases; heuristic-based methods have been proposed to overcome scalability concerns.

An LP model comprises a set of decision variables (possibly constrained to assume only integer values), a set of linear constraints, and an objective function. Constraints and objective function are expressed as (linear) inequalities over the decision variables. The cyclic executive schedule can be modeled as an instance of a 0/1 optimization, as the sought solution will model whether or not a task is mapped to a given computational element. Intuitively, the objective function aims at minimizing the total utilization and failing to find a solution to the LP problem means that the task set is not schedulable under any feasible configuration. Other criteria can be specified in the form of weights to guide mapping decisions. It is worth noting that, in our particular case, we are not interested in finding an optimal solution, but only in proving or disproving the task set schedulability.

To instantiate the task model to the Xavier SoC, and consistently with the investigation conducted in the paper, we consider a sub-set of all the supported CE/TLP configurations $\mathcal{CE}_{Xavier} = \{\texttt{CPU}, \texttt{GPU}^{\texttt{RC}}, \texttt{GPU}^{\texttt{RC-comb}}, \texttt{GPU}^{\texttt{TC}}, \texttt{GPU}^{\texttt{TC-comb}}, \texttt{GPU}^{\texttt{RC+TC}}, \texttt{NVDLA}, \texttt{NVDLA}^{\texttt{comb}}\}$. Here `comb` configurations for the NVDLA and GPU cores hints at the possibility of being constrained to always use the multiple instances of the CE as a block. The set of tasks' timing bounds per configuration is given in input to the ILP as a static bi-dimensional matrix $U[\tau_i \in \mathcal{T}][ce_j \in \mathcal{CE}]$ holding the timing budget of task τ_i when deployed to node ce_j. The main decision variable consists in a bi-dimensional boolean matrix $B[\tau_i \in \mathcal{T}][ce_j \in \mathcal{CE}]$ representing whether τ_i is deployed to ce_j. Accordingly, the objective function would consist in minimizing the cumulative utilization, $\sum_{\tau_i \in \mathcal{T},\ ce_j \in \mathcal{CE}} U[\tau_i][ce_j] * B[\tau_i][ce_j]$. A set of LP constraints has been defined to guarantee a task can only be mapped to one CE/TLP (we assume tasks cannot be deployed to multiple CEs) and to enforce the maximum utilization on a CE/TLP configuration not to exceed 100% [23]. Constraints also handle the inter-correlations between CE/TLP, such as the fact that while two applications can be mapped to `NVDLA` at the same time, $\texttt{NVDLA}^{\texttt{comb}}$ is a configuration that implies exclusive use of the `NVDLA`.

6.2 Experimental setup

In our experiments we assess the impact that supporting different CE and thread-level parallelism (TLP) may have on the schedulability of several DNN instances on the same platform. We build on the information we derived from the Apollo software to perform a scenario-based evaluation. We used the timing profile of the applications analyzed in Section 5 to derive a predefined set of DNN (or RNN) applications (DNN_{1-5}, RNN_{1-3}), with varying computational and timing requirements under the different CE/TLPs configuration (where the set of CE/TLP configurations matches the one considered in Section 5). Each application is represented as a recurrent task with a worst-case execution time distribution, in the range $[U_{max}, U_{min}]$ milliseconds, which depends on the concrete CE/TLP configuration. The time interval has been derived by applying a ±15% inflation factor to the values observed on the Xavier SoC reported in Figure 8. As observed in Section 5.3 those reference values also factor in contention effects and we assume the timing requirements are not changing depending on the deployment configuration of corunning applications.

Table 4 Utilization distributions for the DNN/RNN types and CE/TLP.

		CPU			GPU				NVDLA	
					GPU^{RC}		GPU^{TC}			
		2	4	6	4	8	4	8	1	2
DNN_1	U_{max}	×	×	×	83.95	49.45	44.85	33.35	20.70	12.65
	U_{min}	×	×	×	62.05	36.55	33.15	24.65	15.30	9.35
DNN_3	U_{max}	×	×	×	27.60	17.25	25.30	17.25	12.65	8.05
	U_{min}	×	×	×	20.40	12.75	18.70	12.75	9.35	5.95
DNN_4	U_{max}	×	×	×	67.85	65.55	81.65	72.45	19.55	11.50
	U_{min}	×	×	×	50.15	48.45	60.35	53.55	14.45	8.50
RNN_1	U_{max}	68.19	34.38	23.80	2.30	2.19	2.19	2.19	-	-
	U_{min}	50.41	25.42	17.60	1.70	1.62	1.62	1.62	-	-
RNN_2	U_{max}	21.05	11.62	8.28	2.19	2.07	2.19	2.07	-	-
	U_{min}	15.56	8.59	6.12	1.62	1.53	1.62	1.53	-	-
RNN_3	U_{max}	6.44	3.57	2.53	1.50	1.38	1.50	1.38	-	-
	U_{max}	4.76	2.64	1.87	1.11	1.02	1.11	1.02	-	-
DNN_5	U_{max}	24.50	13.46	10.35	1.50	1.27	1.50	1.15	4.72	3.45
	U_{max}	18.11	9.95	7.65	1.11	0.94	1.11	0.85	3.49	2.55

For each CE/TLP configuration, we generated 16,000 synthetic task sets under different overall utilization thresholds (with a mechanism similar to UUnifast [22]). Task set were generated by randomly selecting several instances of the diverse DNN/RNN types. The utilization of each DNN (RNN) is drawn from the intervals reported in Table 4 above (values are in milliseconds), which is in turn built on the timing characterization results in Section 5.3. DNN_2, the object detector version working with high resolution images in Figure 8, was not included in the evaluation as it is corresponds to a high-resolution variant of object detection application that is clearly over-demanding for the target platform. We use instead DNN_1, the object detector working with standard resolution images. Still on Table 4, it is also worth noting that applications (DNN1, DNN3, DNN4) could not be scheduled on CPU cores (utilization larger than 100%), and RNNs execution is not supported on NVDLA). This is supported in the ILP model by forcing $B[\tau_i][ce_j] = 0$ for specific combinations.

As commented above, focusing on a single scheduling frame is sufficient to fulfill our evaluation objective. We therefore assumed all applications to fit in the same frame, with a reference size of 100ms.

6.3 Schedulability results

We used our LP formulation to assess the schedulability of the task sets under specific CE/TLP configurations. All DNNs (RNNs) are required to run concurrently on the same system, as observed in the case of Apollo. A task set is considered to be infeasible if the LP problem admits no solution. We consider different CE/TLP setting, ranging from single-CE configurations (CPU, GPU^{RC}, GPU^{TC}, and NVDLA only), to mixture configuration, up to the most flexible setting where all CE/TLP configurations are supported. The experiments aim at confirming that being able to configure and exploit different computing elements with different task-level parallelism is a fundamental enabler for successfully deploying multiple DNN variants on the same system. We asses how support for different CE/TLP can be leveraged to sustain the schedulability of systems that would have been not schedulable otherwise. Also, when a system admits multiple feasible schedules, the ILP could be also instructed to identify, among the existing feasible CE/TLP configuration, the one satisfying a predefined criterion, such as maximizing performance.

In order to analyze the benefits of our neural network variant proposal, we use, as a baseline reference, single-CE setups, where only one CE is exploited. We create several scenarios in which an increasing subset of all CEs are used (CPU, GPU^{RC}, GPU^{TC}, NVDLA). In each scenario, the utilization thresholds considered for the experiments are computed on the reference utilization of the CE providing the highest performance.

The scenarios we addressed are the following:

- `nvdla+gpu_rc+gpu_tc+cpu`: takes $\text{NVDLA}^{\text{comb}}$ as reference highest-performance CE, and considers the CE/TLP configurations CPU, GPU^{TC}, $\text{GPU}^{\text{RC+TC}}$, $\text{NVDLA}^{\text{comb}}$, NVDLA;
- `gpu_tc+gpu_rc+cpu`: takes GPU^{TC} as reference highest-performance CE, and considers the CE/TLP configurations CPU, GPU^{TC}, $\text{GPU}^{\text{RC+TC}}$;
- `gpu_rc+cpu`: only uses CPU and GPU^{RC}, with the latter being the highest-performing CE.

This approach lets us assess our variants approach under different scenarios with increasing number of supported CEs, each with its specific performance characteristics. Additionally, we assess the flexibility of considering all the units of the highest-performing CE as a single element with their combined performance ($\text{NVDLA}^{\text{comb}}$) versus providing the scheduler the flexibility to allocate NN instances to independent CE units (NVDLA).

As explained in Section 6.2, a large set of workloads with different NN instances has been generate for each scenario, using the cumulative utilization relative to the highest-performing CE as a threshold. In all scenarios we considered such threshold to vary in 100% to 400% utilization over the scheduling interval.

NVDLA. Figure 10 shows the ratio of feasible task sets under the considered utilization thresholds (relative to NVDLA) and CE/TLP configurations in the `nvdla+gpu_rc+gpu_tc+cpu` scenario. Under 100% NVDLA utilization, the NVDLA alone can always schedule the task set: both in the $\text{NVDLA}^{\text{comb}}$ and NVDLA setups we observed a 100% success ratio. This is obviously the case for $\text{NVDLA}^{\text{comb}}$, as it is the scenario used to compute the utilization threshold. But it also normally holds for two separate instances of NVDLA as the combined use of the $\text{NVDLA}^{\text{comb}}$ does not necessarily exploit full parallelism. Clearly, with increasing utilization, $\text{NVDLA}^{\text{comb}}$ cannot schedule any workload. NVDLA instead still exhibits a high success ratio at 120% utilization, that only falls rapidly at 140% and becomes zero after 160%. This is explained by the fact that the utilization is relative to the *combined* use of NVDLA which is not providing exactly double performance when compared to a single NVDLA instance.

Analyzing the benefits of our variants approach, we can see that enabling the use of other CEs allows to sustain the execution of all NN instances (100% success ratio) for loads up

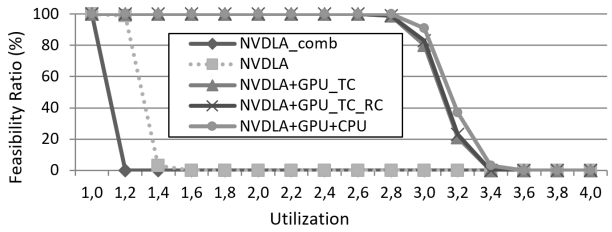


Figure 10 Percentage of schedulable workload when the NVDLA is the highest-performance CE.

Table 5 Average DNN/RNNs per workload in `nvdla+gpu_rc+gpu_tc+cpu` scenarios.

	$\texttt{NVDLA}^{\texttt{comb}}$	NVDLA	NVDLA $\texttt{GPU}^{\texttt{TC}}$	NVDLA $\texttt{GPU}^{\texttt{TC+RC}}$	NVDLA $\texttt{GPU}^{\texttt{TC+RC}}$ CPU
1.0	12.16	12.16	12.16	12.16	12.16
1.2	×	14.66	14.66	14.66	14.66
1.4	×	16.50	17.23	17.23	17.23
1.6	×	×	19.83	19.83	19.83
1.8	×	×	22.46	22.46	22.46
2.0	×	×	24.93	24.93	24.93
2.2	×	×	27.56	27.56	27.56
2.4	×	×	30.13	30.13	30.13
2.6	×	×	32.63	32.63	32.63
2.8	×	×	35.29	35.29	35.24
3.0	×	×	38.46	38.31	38.13
3.2	×	×	43.30	43.22	42.72
3.4	×	×	×	×	49.50

to 2.8, significantly beyond what is observed with NVDLAs only. In between 2.8 and 3.4, the flexibility of CE/TLP deployment is exploited at most, allowing to successfully schedule some task sets. The average numbers of NN-base functionalities successfully scheduled under the considered workloads and CE/TLP configurations are reported in Table 5. Within the feasibility region, all scenarios behaves quite similarly as the average task set population grows as long as the computational load increases. Still within the feasibility region, the average number of instances does not increase when adding more CEs. The only minimal variation happens at 140% utilization, where enabling the GPU allows for one additional NN-based functionality to be successfully deployed in the average case. When the NVDLAs are saturated, the GPU elements alone are capable of providing up to 340% utilization (NVDLA-defined) and changing the GPU configuration (enabling regular CUDA cores) or introducing the CPUs is slightly affecting both schedulability and number of allocated DNNs.

GPU Tensor cores. The ratio of feasible task sets under the `gtc+grc+cpu` scenario is reported in Figure 11. The considered utilization thresholds are relative to the use of 8 $\texttt{GPU}^{\texttt{TC}}$ as a block. Similarly to the NVDLA, the more flexible configuration, where GPU cores are used as two separate clusters, guarantees an improved schedulability ratio.

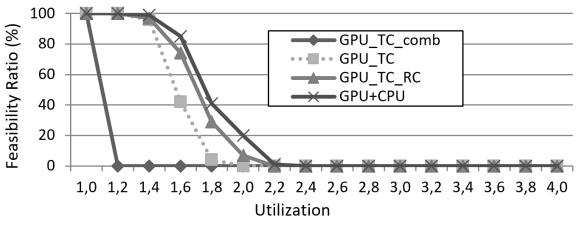


Figure 11 Percentage of schedulable workload when $\texttt{GPU}^{\texttt{TC}}$ is the highest-performance CE.

Table 6 Average DNN/RNNs per workload in `gpu_tc+gpu_rc+cpu` scenarios.

	$\texttt{GPU}^{\texttt{TC}-\texttt{comb}}$	$\texttt{GPU}^{\texttt{TC}}$	$\texttt{GPU}^{\texttt{TC+RC}}$	$\texttt{GPU}^{\texttt{TC+RC}}$ CPU
1.0	10.52	10.52	10.52	10.52
1.2	×	11.27	11.27	11.27
1.4	×	11.55	11.58	11.60
1.6	×	11.41	11.72	12.45
1.8	×	11.33	11.90	12.47
2.0	×	×	13.00	14.10
2.2	×	×	×	9.00

The improvement in terms of schedulability is even larger than in the NVDLA case, as the flexible use of the GPU allows to schedule almost 80% of the task sets even under a 150% workload. When other CEs are enabled, as suggested by our approach, the schedulability ratio further improves and reaches 85% at 160% utilization. It is interesting to note the performance improvement obtained by moving from using only $\texttt{GPU}^{\texttt{TC}}$ as two independent clusters to using potentially both $\texttt{GPU}^{\texttt{TC}}$ and $\texttt{GPU}^{\texttt{RC}}$. In fact, one would expect regular cores not to bring any improvement over the Tensor cores scenario, being the Tensor a more advanced accelerator than regular GPU cores. However, while being more advanced, Tensor cores are also more specialized and their use can be counter-productive for generic applications, as can be also observed in Table 4. Exploiting the CPU, instead, allows a comparatively smaller increase

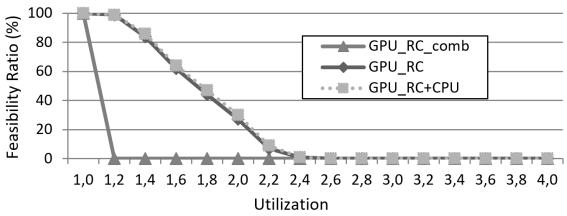


Figure 12 Percentage of schedulable workload when $\texttt{GPU}^{\texttt{RC}}$ is the highest-performance CE.

Table 7 Average DNN/RNNs per workload in `gpu_rc+cpu` scenarios.

	$\texttt{GPU}^{\texttt{RC}-\texttt{comb}}$	$\texttt{GPU}^{\texttt{RC}}$	$\texttt{GPU}^{\texttt{RC}}$ CPU
1.0	9.59	9.59	9.59
1.2	×	10.54	10.54
1.4	×	11.35	11.45
1.6	×	11.96	12.08
1.8	×	12.49	12.65
2.0	×	12.88	13.39
2.2	×	14.01	14.71
2.4	×	18.00	18.35

in computational power, as expected. The average numbers of NN-based functionalities successfully scheduled under the considered workloads and CE/TLP configurations (see Table 6) show substantially similar values for all configurations. The configuration using Tensor cores in clusters of four shows slightly different values than those observed when enabling the regular cores and the CPU. Similarly to the NVDLA scenario, a flexible use of the $\texttt{GPU}^{\texttt{TC}}$ alone allows sustaining up to 200% utilization. Again, introducing the CPUs is not affecting schedulability and is not allowing a larger number of DNN instances.

GPU Regular cores. As final step in our incremental evaluation, we assess the benefits of our approach in an CE/TLP configuration where only the CPU and the GPU regular cores are made available. In this case, the benefit of the flexible approach $\texttt{GPU}^{\texttt{RC}}$ over the combined $\texttt{GPU}^{\texttt{RC}-\texttt{comb}}$ is remarkable. Conversely, the benefit offered by enabling additional CEs is less consistent, when compared to the flexible use of the reference CE. The reason is that regular GPU cores do not seem to be able to support a good degree of parallelism for NN-based functionalities, as confirmed by relatively close performance between using 4 or 8 GPU cores in Table 4. Enabling the use of CPUs only makes a negligible difference in the success ratio, which is explained by the relatively small increase in computational power provided by the CPU cores. The average number of NN-based functionalities scheduled under these configurations (see Table 7), confirms the trend observed for the NVDLA and Tensor cores scenarios. The number of scheduled instances increases with the utilization. Only few DNN instances are added in the average after enabling the use of CPUs.

7 Related Works

DL techniques are increasingly used in critical domains for they deliver substantially more precise functional results compared to other approaches. GPUs are being considered for the execution of DL software because of their capability of performing massively-parallel general-purpose computations and efficiency supporting DL libraries [21]. However, the use of GPUs in CRTES, such as vehicles, brings plenty of challenges for safety [14, 40] and timing. The latter, which can be categorized into three groups, have been addressed by different works: (i) some works focus on the implications of GPUs in the real-time properties of the system, (ii) others aim at improving utilization and efficiency of the existing DL and machine vision software, and (iii) other works propose low-level modifications to support DL such as scheduling algorithms or hardware support.

Research on the real-time properties of GPUs has been conducted for almost a decade. Initial works focused on scheduling proposals for the special timing behavior of GPUs, which is based on interrupts [32], and deal with their non-preemptive nature, which requires task synchronization [34]. Multiple CPU-GPU allocation strategies have been considered in [33], where the authors evaluate different partitioning and clustering schemes to enable sharing multiple instances of the same GPU across multiple cores. In our work we deal with a

heterogeneous set of accelerators, and GPU regular/Tensor cores. We consider a set of diverse parallel tasks that can be scheduled under varying TLP through multiple CPUs, GPU SMs, as well as on other specialized accelerators; we study how their execution requirements varies depending on the computing element they are scheduled on. More recent works have focused on exposing undocumented or mis-documented features of NVIDIA GPUs and their benchmarking [15, 42, 39]. Moreover, [42] is the first real-time paper evaluating NVIDIA's MPS system, which allows multiple processes to execute kernels concurrently in the GPU, containing their SM usage, which is an essential feature for our work. Similarly to our work, [41] considers fine-grained vision-related schedulable entities that can be executed on CPU or GPU, but it does not consider several accelerators beyond the GPU's SMs.

Authors in [43] apply sensor fusion and propose a supervised scheduling algorithm for multiple DNN layers, considering each one as a separate dynamically schedulable entity on a GPU. Similarly to our work, the proposed approach focuses on multiple DNN instances. The focus, however, is limited to a single computational element and does not include the use of multiple elements and thread-level parallelism configurations. Bateni et al. [20] proposed *ApNet*, an approximation-aware real-time neural network, to guarantee that DNN workloads meet their deadlines by using an efficient approximation. Despite their proposal can incur some accuracy loss, it can guarantee the timing predictability. Our work is orthogonal to the ApNet and applying both approaches can further improve resource utilization and performance. In another work, Bateni et al. [21] proposed *Predjoule*, which is a timing predictable energy optimization framework. Predjoule targets DNN workloads and guarantees the latency and energy efficiency of such workloads. We believe that this work can be extended to support various hardware resources and, in combination with our work, could provide better improvements in latency and energy consumption.

Capodieci et al. [24] presented a real-time scheduler for GPU activities on SoC systems such as NVIDIA Jetson TX2. They implement and test Earliest Deadline First (EDF) for GPU tasks, which is enhanced with a Constant Bandwith Server (CBS) based timing isolation mechanism. On the contrary, our work allows the co-scheduling of different computing resources such as CPU cores, GPU cores and Tensor cores, and DL accelerators.

Overall, to the best of our knowledge, this is the first work to study the performance variability of diverse DNN/RNN variants with different computing elements and TLP setups in the Xavier SoC. Also, we exploit a LP model of a heterogeneous static scheduler to assess the capability of the platform to sustain the execution of multiple DNN/RNN instances.

8 Conclusions

As the number of DNN/RNN instances running in parallel continues to increase in future AD systems, so does the ability to exploit the heterogeneous computing elements in modern computing SoCs. In this paper, in support to the first claim, we have analyzed the neural networks concurrently running in the Apollo AD and the current projections in their number. To sustain the latter claim, instead, we have created distinct variants of the different neural-network libraries used in Apollo. Our results show high diversity in the performance obtained by each variant in each of the computing elements of the Jetson AGX Xavier. This diversity provides an opportunity for exploiting the scheduling strategy to simultaneously deploy multiple NN-based instances on the same platform. We used an LP formulation for a multicore cyclic executive scheduler to demonstrate the performance increase potentially enabled by different heterogeneous computing elements, and to show how this allows deploying multiple advanced NN-based functionalities on the same SoC.

References

1 Implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. URL: http://docs.nvidia.com/cuda/cublas/.

2 Intel® GO™ Automated Driving Solution Product Brief. URL: https://www.intel.es/content/dam/www/public/us/en/documents/platform-briefs/go-automated-accelerated-product-brief.pdf.

3 NVIDIA DRIVE PX. Scalable supercomputer for autonomous driving. URL: http://www.nvidia.com/object/drive-px.html.

4 QUALCOMM Snapdragon 820 Automotive Processor. URL: https://www.qualcomm.com/products/snapdragon/processors/820-automotive.

5 RENESAS R-Car H3. URL: https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html.

6 TensorRT: A platform for high-performance deep learning inference. URL: https://developer.nvidia.com/tensorrt.

7 TensorRT Support Matrix. URL: https://docs.nvidia.com/deeplearning/sdk/tensorrt-support-matrix/index.html.

8 AUTOMATED DRIVING, Levels of driving automation are deined in new SAE International standard J3016., 2014. URL: https://www.sae.org/standards/content/j3016_201609/.

9 APOLLO, an open autonomous driving platform., 2018. URL: http://apollo.auto/.

10 Deep Learning SDK Documentation, 2018. URL: https://docs.nvidia.com/deeplearning/sdk/tensorrt-archived/tensorrt-504/tensorrt-support-matrix/index.html.

11 Self-driving Safety Report, 2018. URL: https://www.nvidia.com/en-us/self-driving-cars/safety-report/.

12 Tensor Core, The Next Generation of Deep Learning., 2018. URL: https://www.nvidia.com/en-us/data-center/tensorcore/.

13 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/.

14 Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain. *IEEE Micro*, 38(6):46–55, 2018.

15 Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

16 ARINC. *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc, 1997.

17 ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade, 2015. URL: https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php.

18 AUTOSAR. *Specification of RTE Software - AUTOSAR CP Release 4.3.1*, 2017.

19 Sanjoy K. Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *Journal of Scheduling*, pages 1–15, 2018.

20 Soroush Bateni and Cong Liu. ApNet: Approximation-Aware Real-Time Neural Network. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

21 Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. PredJoule: A Timing-Predictable Energy Optimization Framework for Deep Neural Networks. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

22 Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1):129–154, 2005.

23 Alan Burns, C Deutschbein, Thomas David Fleming, and S Baruah. Multi-core Cyclic Executives for Safety-Critical Systems. *Dependable Software Engineering Theories, Tools and Application*, 172:94–109, 2017.

24 Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based Scheduling for GPU with Preemption Support. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

25 Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *IEEE Emerging Technologies and Factory Automation (ETFA)*, 2017.

26 Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint*, 2014.

27 François Chollet. Keras, 2015. URL: `https://github.com/fchollet/keras`.

28 Tesla Corp. Tesla Autopilot, 2018. URL: `https://www.tesla.com/autopilot`.

29 Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering (CiSE)*, 5(1):46–55, 1998.

30 Nachiket Deo and Mohan M Trivedi. Looking at the Driver/Rider in Autonomous Vehicles to Predict Take-Over Readiness. *arXiv preprint*, 2018.

31 Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the AURIX^{TM} TC27x. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

32 Glenn A. Elliott and James H. Anderson. Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

33 Glenn A. Elliott and James H. Anderson. Exploring the Multitude of Real-Time Multi-GPU Configurations. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

34 Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. GPUSync: A Framework for Real-Time GPU Management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

35 Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, 2014.

36 Joël Goossens, Pascal Richard, Markus Lindström, Irina Iulia Lupu, and Frédéric Ridouard. Job Partitioning Strategies for Multiprocessor Scheduling of Real-time Periodic Tasks with Restricted Migrations. In *ACM Real-Time and Network Systems (RTNS)*, 2012.

37 Richard Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, 40:85–103, 1972.

38 Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

39 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alex Berg, and Shige Wang. An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

40 Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Guillem Bernat, and Francisco J Cazorla. Assessing the Adherence of Industrial Autonomous Driving Software to ISO-26262 Guidelines for Software. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2019.

41 Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Shige Wang. Making OpenVX Really "Real Time". In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

42 Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

43 Husheng Zhou, Soroush Bateni, and Cong Liu. S^3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

44 Alex Zyner, Stewart Worrall, and Eduardo Nebot. A Recurrent Neural Network Solution for Predicting Driver Intention at Unsignalized Intersections. *IEEE Robotics and Automation Letters (RA-L)*, 3(3):1759–1764, 2018.

45 Alex Zyner, Stewart Worrall, and Eduardo Nebot. Naturalistic Driver Intention and Path Prediction Using Recurrent Neural Networks. *arXiv preprint*, 2018.